Specialization with Constrained Generalization for Software Model Checking

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] DEC, University 'G. D'Annunzio', Pescara, Italy
{emanuele.deangelis,fioravanti}@unich.it
[2] DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present a method for verifying properties of imperative programs by using techniques based on constraint logic programming (CLP). We consider a simple imperative language, called SIMP, extended with a nondeterministic choice operator and we address the problem of checking whether or not a *safety* property φ (that specifies that an *unsafe* configuration cannot be reached) holds for a SIMP program P. The operational semantics of the language SIMP is specified via an interpreter I written as a CLP program. The first phase of our verification method consists in specializing I with respect to P, thereby deriving a specialized interpreter I_P. Then, we specialize I_P with respect to the property φ and the input values of P, with the aim of deriving, if possible, a program whose least model is a finite set of constrained facts. To this purpose we introduce a novel generalization strategy which, during specialization, has the objecting of preserving the so called branching behaviour of the predicate definitions. We have fully automated our method and we have made its experimental evaluation on some examples taken from the literature. The evaluation shows that our method is competitive with respect to state-of-the-art software model checkers.

1 Introduction

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see [19] for a recent survey).

In this paper we consider a simple imperative language SIMP acting on integer variables, with nondeterministic choice, assignment, conditional, and while-do commands (see, for instance, [29]) and we address the problem of verifying *safety* properties. Basically, a safety property states that when executing a program, an unsafe configuration cannot be reached from any initial configuration. Note that, since we consider programs that act on integer numbers, the problem of deciding whether or not an unsafe configuration is unreachable is in general undecidable.

In order to cope with this undecidability limitation, many program analysis techniques have followed approaches based on *abstraction* [4], by which the concrete data domain is mapped to an abstract domain so that reachability is preserved, that is, if a concrete configuration is reachable, then the corresponding abstract configuration is reachable. By a suitable choice of the abstract domain one can design reachability algorithms that terminate and, whenever they prove that an abstract unsafe configuration is unreachable from an abstract initial configuration, then the program is proved to be safe (see [19] for a general abstract reachability algorithm). Notable abstractions are those based on convex polyhedra, that is, conjunctions of linear inequalities (also called *constraints* here).

Due to the use of abstraction, the reachability of an abstract unsafe configuration does not necessarily imply that the program is indeed unsafe. It may happen that the abstract reachability algorithm produces a *spurious counterexample*, that is, a sequence of configurations leading to an abstract unsafe configuration which does not correspond to any concrete computation. When a spurious counterexample is found, *counterexample-guided abstraction refinement* (CEGAR) automatically refines the abstract domain so that a new run of the abstract reachability algorithm rules out the counterexample [1,3,30]. Clearly, the CEGAR technique may not terminate because an infinite number of spurious counterexamples may be found. Thus, in order to improve the termination behaviour of that technique, several more sophisticated refinement strategies have been proposed (see, for instance, [14,16,20,32]).

In this paper in order to improve the termination of the safety verification process, we propose a technique based on the *specialization of constraint logic programs*. Constraint Logic Programming (CLP) has been shown to be very suitable for the analysis of imperative programs, because it provides a very convenient way of representing symbolic program executions and also, by using constraints, program invariants (see, for instance, [16,18,27,28]). Program specialization is a program transformation technique which, given a program P and a portion in_1 of its input data, returns a specialized program P_s that is equivalent to P in the sense that when the remaining portion in_2 of the input of P is given, then $P_s(in_2) = P(in_1, in_2)$ [12,21,22]. The specialization of CLP programs has been proposed in [27] as a pre-processing phase for program analysis. This analysis is done in various steps. First, the semantics of an imperative language is provided by means of a CLP program which defines the interpreter I of that language, and then, program I is specialized with respect to the program P whose safety property should be checked. The result of this specialization is a CLP program I_P and, since program specialization preserves semantic equivalence, we can analyze I_P for proving the properties of P.

Similarly to [27], also the technique proposed in this paper produces a specialized interpreter I_P. However, instead of applying program analysis techniques, we further specialize I_P with respect to the property characterizing the input values of P (that is, the precondition of P), thereby deriving a new program I'_P. The effect of this further specialization is the modification of the structure of the program I_P and the explicit addition of new constraints that denote invariants of the computation. Through various experiments we show that by exploiting these invariants, the construction of the least model of the program I'_P terminates in many interesting cases and, thus, it is possible to verify safety properties by simply inspecting that model.

An essential ingredient of program specialization are the *generalization steps*, which introduce new predicate definitions representing invariants of the program executions. Generalizations can be used to enforce the termination of program specialization (recall that termination occurs when no new predicate definitions are generated) and, in this respect, they are similar to the widening operators used in static program analysis [4,5]. One problem encountered with generalizations is that sometimes they introduce predicate definitions which are too general, thereby making specialization useless. In this paper we introduce a new generalization strategy, called the *constrained generalization*, whose objective is indeed to avoid the introduction of new predicate definitions that are too general.

The basic idea of the constrained generalization is related to the branching behaviour of the unfolding steps, as we now indicate. Given a sequence of unfolding steps performed during program specialization, we may consider a symbolic evaluation tree made out of clauses, such that every clause has as children the clauses which are generated by unfolding that clause. Suppose that a clause γ has n children which are generated by unfolding using clauses $\gamma_1, \ldots, \gamma_n$, and suppose that during program specialization we have to generalize clause γ. Then, we would like to perform this generalization by introducing a new predicate definition, say δ, such that by unfolding clause δ, we get again, if possible, n children and these children are due to the same clauses $\gamma_1, \ldots, \gamma_n$.

Since in this generalization the objective of preserving, if possible, the branching structure of the symbolic evaluation tree, is realized by adding extra constraints to the clause obtained after a usual generalization step (using, for instance, the widening operator [4] or the convex-hull operator [5]), we call the generalization proposed in this paper *a constrained generalization*. Similar proposals have been presented in [2,15] and in Section 7 we will briefly compare those proposals with ours.

The paper is organized as follows. In Section 2 we describe the syntax of the SIMP language and the CLP interpreter which defines the operational semantics of that language. In Section 3 we outline our software model checking approach by developing an example taken from [14]. In Sections 4 and 5 we describe our strategy for specializing CLP programs and, in particular, our novel constrained generalization technique. In Section 6 we report on some experiments we have performed by using a prototype implementation based on the MAP transformation system [26]. We also compare the results we have obtained using the MAP system with the results we have obtained using state-of-the-art software model checking systems such as ARMC [28], HSF(C) [13], and TRACER [17]. Finally, in Section 7 we discuss the related work and, in particular, we compare our method with other existing methods for software model checking.

2 A CLP Interpreter for a Simple Imperative Language

The syntax of our language SIMP, a C-like imperative language, is defined by using: (i) the set *Int* of integers, ranged over by n, (ii) the set {`true`, `false`} of booleans, and (iii) the set *Loc* of locations, ranged over by x. We have also the following derived sets: (iv) *Aexpr* of arithmetic expressions, (v) *Bexpr* of boolean expressions, (vi) *Test* of tests, and (vii) *Com* of commands. The syntax of our language is as follows.

$Aexpr \ni a ::= n \mid x \mid a_0 \; aop \; a_1$
$Bexpr \ni b ::= \texttt{true} \mid \texttt{false} \mid a_0 \; rop \; a_1 \mid \texttt{!}\, b \mid b_0 \; bop \; b_1$
$Test \quad \ni t ::= \texttt{nd} \mid b$
$Com \quad \ni c ::= \texttt{skip} \mid x \,\texttt{=}\, a \mid c_0\,\texttt{;}\,c_1 \mid \texttt{if (}\,t\,\texttt{) \{}\,c_0\,\texttt{\} else}\;c_1 \mid \texttt{while (}\,t\,\texttt{) \{}\,c\,\texttt{\}} \mid \texttt{error}$

where the arithmetic operator *aop* belongs to {`+`, `-`, `*`}, the relational operator *rop* belongs to {`<`, `<=`, `==`}, and the boolean operator *bop* belongs to {`&&`, `||`}. The constant `nd` denotes the nondeterministic choice and `error` denotes the error command. The other symbols should be understood as usual in C. We will write `if (`t`) {`c_0`}`, instead of `if (`t`) {`c_0`} else skip`.

Now we introduce a CLP program which defines the interpreter of our SIMP language. We need the following notions.

A *state* is a function from *Loc* to *Int*. It is denoted by a list of CLP terms, each of which is of the form `bn(loc(X),V)`, where `bn` is a binary constructor binding the location `X` to the value of the CLP variable `V`. We assume that the set of locations used in every command is fixed and, thus, for every command, the state has a fixed, finite length. We have two predicates operating on states: (i) `lookup(loc(X),S,V)`, which holds iff the location `X` stores the value `V` in the state `S`, and (ii) `update(loc(X),V,S1,S2)`, which holds iff the state `S2` is equal to the state `S1`, except that the location `X` stores the value `V`.

We also have the predicates `aev(A,S,V)` and `bev(B,S)`, for the evaluation of arithmetic expressions and boolean expressions, respectively. `aev(A,S,V)` holds iff the arithmetic expression `A` in the state `S` evaluates to `V`, and `bev(B,S)` holds iff the boolean expression `B` holds in the state `S`. A test `T` in a state `S` is evaluated via the predicate `tev(T,S)` defined as follows: (i) for all states `S`, *both* `tev(nd,S)` and `tev(not(nd),S)` hold (and in this sense `nd` denotes the nondeterministic choice), and (ii) for all boolean expressions `B`, `tev(B,S)` holds iff `bev(B,S)` holds.

A command c is denoted by a term built out of the following constructors: `skip` (nullary), `asgn` (binary) for the assignment, `comp` (binary) for the composition of command, `ite` (ternary) for the conditional, and `while` (binary) for the while-do. The operator '`;`' associates to the right. Thus, for instance, the command c_0`;`c_1`;`c_2 is denoted by the term `comp(c0,comp(c1,c2))`.

A *configuration* is a pair of a command and a state. A configuration is denoted by the term `cf(c,s)`, where `cf` is a binary constructor which takes as arguments the command `c` and the state `s`. The interpreter of our SIMP language, adapted from [29], is defined in terms of a *transition relation* that relates an old configuration to either a new configuration or a new state. That relation is denoted by the predicate `tr` whose clauses are given below. `tr(cf(C,S),cf(C1,S1))` holds iff the execution of the command `C` in the state `S` leads to the new configuration `cf(C1,S1)`, and `tr(cf(C,S),S1)` holds iff the execution of the command `C` in the state `S` leads to the new state `S1`.

```
tr(cf(skip,S), S).
tr(cf(asgn(loc(X),A),S),S1) :- aev(A,S,V), update(loc(X),V,S,S1).
tr(cf(comp(C0,C1),S), cf(C1,S1)) :- tr(cf(C0,S),S1).
tr(cf(comp(C0,C1),S), cf(comp(C0',C1),S')) :- tr(cf(C0,S), cf(C0',S')).
tr(cf(ite(T,C0,C1),S), cf(C0,S)) :- tev(T,S).
tr(cf(ite(T,C0,C1),S), cf(C1,S)) :- tev(not(T),S).
tr(cf(while(T,C),S),cf(ite(T,comp(C,while(T,C)),skip),S)).
```

A state `s` is said to be *initial* if *initProp* holds in `s`. A configuration is said to be *initial* if its state is initial. A configuration is said to be *unsafe* if its command is `error`.

Now, we introduce a CLP program, called R, that by using a bottom-up evaluation strategy, performs in a backward way the reachability analysis over configurations. Program R checks whether or not an unsafe configuration is reachable from an initial configuration, by starting from the unsafe configurations. The semantics of program R is given by its least model, denoted $M(R)$.

Definition 1 (Reachability Program). *Given a boolean expression initProp holding in the initial states and a command com, the reachability program R is made out of the following clauses:*

```
unsafe :- initConf(X), reachable(X).
reachable(X) :- unsafeConf(X).          % unsafe configurations are reachable
reachable(X) :- tr(X,X1), reachable(X1).
initConf(cf(com,S)) :- bev(initProp,S).% initProp holds in the initial state S
unsafeConf(cf(error,S)).  % the error command defines an unsafe configuration
```

together with the clauses for the predicates `tr` *and* `bev` *and the predicates they depend upon. In the above clauses for R the terms* `initProp` *and* `com` *denote initProp and com, respectively. We will say that com is safe with respect to initProp (or com is safe, for short) iff* `unsafe` $\notin M(R)$.

3 Specialization-Based Software Model Checking

In this section we outline the method for software model checking we propose. By means of an example borrowed from [14], we argue that program specialization can prove program safety in some cases where the CEGAR method (as implemented in ARMC [28]) does not work.
Let the property *initProp* which characterizes the initial states be:

```
x==0 && y==0 && n>=0
```

and the SIMP command *com* be:

```
while (x<n) { x = x+1; y = y+1 };
while (x>0) { x = x-1; y = y-1 };
if (y>x) error
```

We want to prove that *com* is safe with respect to *initProp*, that is, there is no execution of *com* with input values of `x`, `y`, and `n` satisfying *initProp*, such that the `error` command is executed. As shown in Table 1 of Section 6, CEGAR fails to prove this safety property, because an infinite set of counterexamples is generated (see the entry '∞' for Program *re*1 in the ARMC column).

By applying the specialization-based software model checking method we propose in this paper, we will be able to prove that *com* is indeed safe. As indicated in Section 2, we have to show that `unsafe` $\notin M(R)$, where R is the CLP program of Definition 1, `com` is the term:

```
comp(while(lt(loc(x),loc(n)),
  comp(asgn(loc(x),plus(loc(x),1)), asgn(loc(y),plus(loc(y),1)))),
comp(while(gt(loc(x),0),
  comp(asgn(loc(x),minus(loc(x),1)), asgn(loc(y),minus(loc(y),1)))),
ite(gt(loc(y),loc(x)),error,skip)))
```

and `initProp` is the term:

```
and(eq(loc(x),0), and(eq(loc(y),0), ge(loc(n),0)))
```

Our method consists of the three phases as we now specify.

The Software Model Checking Method
Input: A boolean expression *initProp* characterizing the initial states and a SIMP command *com*.
Output: The answer *safe* iff *com* is safe with respect to *initProp*.

Let R be the CLP program of Definition 1 defining the predicate `unsafe`.
Phase (1): $\mathit{Specialize}_{\texttt{com}}(R, R_{\texttt{com}})$;
Phase (2): $\mathit{Specialize}_{\texttt{initProp}}(R_{\texttt{com}}, R_{Sp})$;
Phase (3): $\mathit{BottomUp}(R_{Sp}, M_{Sp})$;
Return the answer *safe* iff `unsafe` $\notin M_{Sp}$.

During Phase (1), by making use of familiar transformation rules (definition introduction, unfolding, folding, removal of clauses with unsatisfiable body, and removal of subsumed clauses [7]), we 'compile away', similarly to [27], the SIMP interpreter by specializing program R with respect to `com`, thereby deriving the following program $R_{\texttt{com}}$ which encodes the reachability relation associated with the interpreter specialized with respect to `com`:

```
1. unsafe :- X=1, Y=1,  N>=1,   new1(X,Y,N).
2. new1(X,Y,N) :- X<N,  X'=X+1, Y'=Y+1, new1(X',Y',N).
3. new1(X,Y,N) :- X>=1, X>=N,   X'=X-1, Y'=Y-1, new2(X',Y',N).
4. new1(X,Y,N) :- X=<0, X<Y,    X>=N.
5. new2(X,Y,N) :- X>=1, X'=X-1, Y'=Y-1, new2(X',Y',N).
6. new2(X,Y,N) :- X=<0, X<Y.
```

The specialization of Phase (1) is said to perform 'the removal of the interpreter'. Note that: (i) the two predicates `new1` and `new2` correspond to the two while-do commands occurring in *com*, and (ii) the assignments and the conditional occurring in *com*, do not occur in $R_{\texttt{com}}$ because by unfolding they have been replaced by suitable constraints relating the values of `X` and `Y` (that is, the old values of the SIMP variables `x` and `y`) to the values of `X'` and `Y'` (that is, the new values of those variables `x` and `y`).

Unfortunately, the program $R_{\texttt{com}}$ is not satisfactory for showing safety, because the bottom-up construction of the least model $M(R_{\texttt{com}})$ does not terminate. The top-down evaluation of the `unsafe` query in $R_{\texttt{com}}$ does not terminate either. Then, in Phase (2) we specialize program $R_{\texttt{com}}$ with respect to the property `initProp`, thereby deriving the specialized program R_{Sp}. During this Phase (2) the constraints occurring in the definitions of `new1` and `new2` are generalized according to a suitable generalization strategy based both on widening [4,8,11] and on the novel constrained generalization strategy we propose in this paper. Suitable new predicate definitions will be introduced during this Phase (2), so that at Phase (3) we can construct the least model M_{Sp} of the derived program R_{Sp} by using a bottom-up evaluation procedure. We will show that, in our example, the construction of the least model M_{Sp} terminates and we can prove the safety of the command *com* by showing that the atom `unsafe` does not belong to that model.

Phase (2) of our method makes use of the same transformation rules used during Phase (1), but those rules are applied according to a different strategy, whose effect is the propagation of the constraints occurring in $R_{\texttt{com}}$.

We start off by introducing the following definition:

7. `new3(X,Y,N) :- X=1, Y=1, N>=1, new1(X,Y,N).`

and then folding clause 1 by using this clause 7. We get the folded clause:

1.f `unsafe:- X=1, Y=1, N>=1, new3(X,Y,N).`

We proceed by following the usual unfold-definition-fold cycle of the specialization strategies [8,11]. Each new definition introduced during specialization determines a new node of a tree, called *DefsTree*, whose root is clause 7, which is the first definition we have introduced. (We will explain below how the tree *DefsTree* is incrementally constructed.) Then, we unfold clause 7 and we get:

8. `new3(X,Y,N) :- X=1, Y=1, N>=2, X'=2, Y'=2, new1(X',Y',N).`

9. `new3(X,Y,N) :- X=1, Y=1, N=1,  X'=0, Y'=0, new2(X',Y',N).`

Now, we should fold these two clauses. Let us deal with them, one at the time, and let us first consider clause 8. In order to fold clause 8 we consider a definition, called the *candidate definition*, which is of the form:

10. `new4(X,Y,N) :- X=2, Y=2, N>=2, new1(X,Y,N).`

The body of this candidate definition is obtained by projecting the constraint in clause 8 with respect to `X'`, `Y'`, and `N`, and renaming the primed variables to unprimed variables. Since in *DefsTree* there is *an ancestor definition*, namely the root clause 7, with the predicate `new1` in the body, we apply the *widening operator*, introduced in [11], to clause 7 and clause 10, and we get the definition:

11. `new4(X,Y,N) :- X>=1, Y>=1, N>=1, new1(X,Y,N).`

(Recall that the widening operation of two clauses $c1$ and $c2$, after replacing every equality `A=B` by the equivalent conjunction `A>=B, A=<B`, keeps the atomic constraints of clause $c1$ which are implied by the constraint of clause $c2$.)

At this point, we do *not* introduce clause 11 (as we would do if we perform a usual generalization using widening alone, as indicated in [8,11]), but we apply our *constrained generalization*, which imposes the addition of some extra constraints to the body of clause 11, as we now explain.

With each predicate `newk` we associate a set of constraints, called the *regions for* `newk`, which are all the *atomic constraints* on the unprimed variables (that is, the variables in the heads of the clauses) occurring in any one of the clauses for `newk` in program $R_{\texttt{com}}$. Then, we add to the body of the generalized definition obtained by widening, say `newp(...) :- c, newk(...)`, (clause 11, in our case), all *negated regions for* `newk` which are implied by `c`.

In our example, the regions for `new1` are: `X<N`, `X>=1`, `X>=N`, `X=<0`, `X<Y` (see clauses 2, 3, and 4) and the negated regions are, respectively: `X>=N`, `X<1`, `X<N`, `X>0`, `X>=Y`. The negated regions implied by the constraint `X=2, Y=2, N>=2`, occurring in the body of the candidate clause 10, are: `X>0` and `X>=Y`.

Thus, instead of clause 11, we introduce the following clause 12 (we wrote neither `X>0` nor `X>=1` because those constraints are implied by `X>=Y, Y>=1`):

12. `new4(X,Y,N) :- X>=Y, Y>=1, N>=1, new1(X,Y,N).`

and we say that clause 12 has been obtained by constrained generalization from clause 10. Clause 12 is placed in *DefsTree* as a child of clause 7, as clause 8 has been derived by unfolding clause 7. By folding clause 8 using clause 12 we get:

8.f `new3(X,Y,N) :- X=1, Y=1, N>=2, X'=2, Y'=2, new4(X',Y',N).`

Now, it remains to fold clause 9 and in order to do so, we consider the following candidate definition:

13. `new5(X,Y,N) :- X=0, Y=0, N=1, new2(X,Y,N).`

Clause 13 is placed in *DefsTree* as a child of clause 7, as clause 9 has been derived by unfolding clause 7. We do not make any generalization of this clause, because no definition with `new2` in its body occurs as an ancestor of clause 13 in *DefsTree*. By folding clause 9 using clause 13 we get:

9.f `new3(X,Y,N) :- X=1, Y=1, N=1, X'=0, Y'=0, new5(X',Y',N).`

Now, we consider the last two definition clauses we have introduced, that is, clauses 12 and 13. First, we deal with clause 12. Starting from that clause, we perform a sequence of unfolding-definition-folding steps similar to the sequence we have described above, when presenting the derivation of clauses 8.f and 9.f, starting from clause 7. During this sequence of steps, we introduce two predicates, `new6` and `new7` (see the definition clauses 16 and 18, respectively), for performing the required folding steps. We get the following clauses:

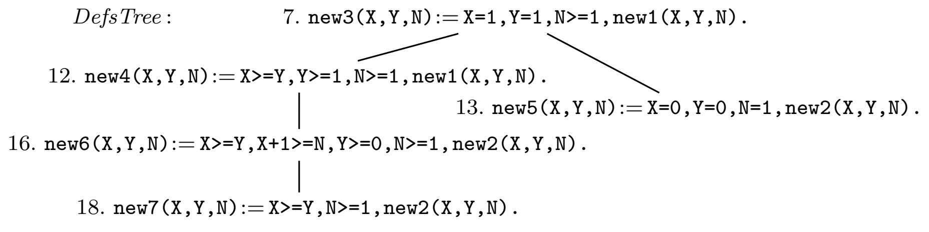
14.f `new4(X,Y,N):-X>=Y, X<N, Y>0, X'=X+1, Y'=Y+1, new4(X',Y',N).`

15.f `new4(X,Y,N):-X>=Y, X>=N, Y>0, N>0, X'=X-1, Y'=Y-1, new6(X',Y',N).`

17.f `new6(X,Y,N):-X>0, X>=Y, X>=N-1, Y>=0, N>0, X'=X-1, Y'=Y-1, new7(X',Y',N).`

19.f `new7(X,Y,N):-X>0, X=<Y, N>0, X'=X-1, Y'=Y-1, new7(X',Y',N).`

The tree *DefsTree* of all the definitions introduced during Phase (2), can be depicted as follows:



DefsTree:
7. new3(X,Y,N):- X=1,Y=1,N>=1,new1(X,Y,N).
12. new4(X,Y,N):- X>=Y,Y>=1,N>=1,new1(X,Y,N).
13. new5(X,Y,N):- X=0,Y=0,N=1,new2(X,Y,N).
16. new6(X,Y,N):- X>=Y,X+1>=N,Y>=0,N>=1,new2(X,Y,N).
18. new7(X,Y,N):- X>=Y,N>=1,new2(X,Y,N).

Then, we deal with clause 13. Again, starting from that clause we perform a sequence of unfolding-definition-folding steps. By unfolding clause 13 w.r.t. `new2` we get an empty set of clauses for `new5`. Then, we delete clause 9.f because there are no clauses for `new5`.

Eventually, we get the program R_{Sp} made out of the following clauses:

```
1.f  unsafe :- X=1, Y=1, N>=1, new3(X,Y,N).
7.1f new3(X,Y,N):-X=1, Y=1, N>=2, X'=2, Y'=2, new4(X',Y',N).
```

together with the clauses 14.f, 15.f, 17.f, and 19.f.

This concludes Phase (2).

Now, we can perform Phase (3) of our method. This phase terminates immediately because in R_{Sp} there are no constrained facts (that is, clauses whose bodies consist of constraints only) and $M(R_{Sp})$ is the empty set.

Thus, `unsafe` $\notin M(R_{Sp})$ and we conclude that the command *com* is safe with respect to *initProp*.

One can verify that if we were to do the generalization steps of Phase (2) using the widening technique alone (without the constrained generalization), we could not derive a program that allows us to prove safety, because during Phase (3) the execution of the *BottomUp* procedure does not terminate.

4 The Specialization Strategy

Phases (1) and (2) of our Software Model Checking method outlined in Section 3 are realized by two applications of a single, general specialization strategy for CLP programs that we now present.

This strategy is an adaptation of the specialization strategies we have presented in [8,11] and, as already mentioned in Section 3, it makes use of the following transformation rules: definition introduction, unfolding, clause removal, and folding. These rules, under suitable conditions, guarantee that the least model semantics is preserved (see, for instance, [7]).

Our general specialization strategy is realized by the following *Specialize* procedure.

Procedure *Specialize*

Input: A CLP program of the form $P \cup \{\gamma_0\}$, where γ_0 is `unsafe` $\leftarrow c, G$.

Output: A CLP program P_S such that `unsafe` $\in M(P \cup \{\gamma_0\})$ iff `unsafe` $\in M(P_S)$.

```
$P_S := \{\gamma_0\}$;   $InDefs := \{\gamma_0\}$;   $Defs := \emptyset$;
while  there exists a clause $\gamma$ in $InDefs$
do  $Unfold(\gamma, \Gamma)$;
    $Generalize\&Fold(Defs, \Gamma, NewDefs, \Delta)$;
    $P_S := P_S \cup \Delta$;  $InDefs := (InDefs - \{\gamma\}) \cup NewDefs$;  $Defs := Defs \cup NewDefs$;
end-while
```

Initially, this procedure considers the clause γ_0 of the form:

$\texttt{unsafe} \leftarrow c, G$

where c is a constraint and G is a goal, and then iteratively applies the following two procedures: (i) the *Unfold* procedure, which uses the unfolding rule and the clause removal rule, and (ii) the *Generalize&Fold* procedure, which uses the definition introduction rule and the folding rule.

The *Unfold* procedure takes as input a clause γ and returns as output a set Γ of clauses derived from γ by one or more applications of the unfolding rule, which consists in: (i) replacing an atom A occurring in the body of a clause by the bodies of the clauses in P whose head is unifiable with A, and (ii) applying the unifying substitution. The first step of the *Unfold* procedure consists in unfolding γ with respect to the leftmost atom in its body. In order to guarantee the termination of the *Unfold* procedure, an atom A is selected for unfolding only if it has not been derived by unfolding a variant of A itself. More sophisticated unfolding strategies can be applied (see [22] for a survey of techniques for controlling unfolding), but our simple strategy turns out to be effective in all our examples. At the end of the *Unfold* procedure, subsumed clauses and clauses with unsatisfiable constraints are removed.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and introduces a set *NewDefs* of *definitions*, that is, clauses of the form $newp(X) \leftarrow d(X), A(X)$, where *newp* is a new predicate symbol, X is a tuple of variables, $d(X)$ is a constraint whose variables are among the ones in X, and $A(X)$ is an atom whose variables are exactly those of the tuple X. Any such definition denotes a set of states X satisfying the constraint $d(X)$. By folding the clauses in Γ using the definitions in *NewDefs* and the definitions introduced during previous iterations of the specialization procedure, the *Generalize&Fold* procedure derives a new set of specialized clauses. In particular, a clause of the form:

$newq(X) \leftarrow c(X), A(X)$

obtained by the *Unfold* procedure, is folded by using a definition of the form:

$newp(X) \leftarrow d(X), A(X)$

if for all X, $c(X)$ implies $d(X)$. This condition is also denoted by $c(X) \sqsubseteq d(X)$, where the quantification 'for all X' is silently assumed. If $c(X) \sqsubseteq d(X)$, we say that $d(X)$ is a *generalization* of $c(X)$. The result of folding is the specialized clause:

$newq(X) \leftarrow c(X), newp(X).$

The specialization strategy proceeds by applying the *Unfold* procedure followed by the *Generalize&Fold* procedure to each clause in *NewDefs*, and terminates when no new definitions are needed for performing folding steps. Unfortunately, an uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions, thereby causing the nontermination of the specialization procedure. In the following section we will define suitable *generalization operators* which guarantee the introduction of finitely many new definitions.

5 Constrained Generalization

In this section we define the generalization operators which are used to ensure the termination of the specialization strategy and, as mentioned in the Introduction, we also introduce *constrained* generalization operators that generalize the constraints occurring in a candidate definition and, by adding suitable extra constraints, have the objective of preventing that the set of clauses generated by unfolding the generalized definition is larger than the set of clauses generated by unfolding the candidate definition. In this sense we say the objective of constrained generalization is to preserve the branching behaviour of the candidate definitions.

Let $\mathcal{C}$ denote the set of all linear constraints. The set $\mathcal{C}$ is the minimal set of constraints which: (i) includes all atomic constraints of the form either $p_1 \leq p_2$ or $p_1 < p_2$, where p_1 and p_2 are linear polynomials with variables $X_1, \ldots, X_k$ and integer coefficients, and (ii) is closed under conjunction (which we denote by ',' and also by '$\wedge$'). An equation $p_1 = p_2$ stands for $p_1 \leq p_2 \wedge p_2 \leq p_1$. The projection of a constraint c onto a tuple X of variables, denoted $\mathit{project}(c, X)$, is a constraint such that $\mathcal{R} \models \forall X\ (\mathit{project}(c, X) \leftrightarrow \exists Y c)$, where Y is the tuple of variables occurring in c and not in X, and $\mathcal{R}$ is the structure of the real numbers.

In order to introduce the notion of a generalization operator (see also [11], where the set $\mathcal{C}$ of all linear constraints with variables $X_1, \ldots, X_k$ has been denoted Lin_k), we need the following definition [6].

Definition 2 (Well-Quasi Ordering $\precsim$). A *well-quasi ordering* (or *wqo*, for short) on a set S is a reflexive, transitive relation $\precsim$ on S such that, for every infinite sequence $e_0 e_1 \ldots$ of elements of S, there exist i and j such that $i < j$ and $e_i \precsim e_j$. Given e_1 and e_2 in S, we write $e_1 \approx e_2$ if $e_1 \precsim e_2$ and $e_2 \precsim e_1$. A wqo $\precsim$ is *thin* iff for all $e \in S$, the set $\{e' \in S \mid e \approx e'\}$ is finite.

The use of a thin wqo guarantees that during the *Specialize* procedure each definition can be generalized a finite number of times only, and thus the termination of the procedure is guaranteed.

The thin wqo *Maxcoeff*, denoted by $\precsim_M$, compares the maximum absolute values of the coefficients occurring in polynomials. It is defined as follows. For any atomic constraint a of the form $p < 0$ or $p \leq 0$, where p is $q_0 + q_1 X_1 + \ldots + q_k X_k$, we define $\mathit{maxcoeff}(a)$ to be $\max\{|q_0|, |q_1|, \ldots, |q_k|\}$. Given two atomic constraints a_1 of the form $p_1 < 0$ and a_2 of the form $p_2 < 0$, we have that $a_1 \precsim_M a_2$ iff $\mathit{maxcoeff}(a_1) \leq \mathit{maxcoeff}(a_2)$.

Similarly, if we are given the atomic constraints a_1 of the form $p_1 \leq 0$ and a_2 of the form $p_2 \leq 0$. Given two constraints $c_1 \equiv a_1, \ldots, a_m$, and $c_2 \equiv b_1, \ldots, b_n$, we have that $c_1 \precsim_M c_2$ iff, for $i = 1, \ldots, m$, there exists $j \in \{1, \ldots, n\}$ such that $a_i \precsim_M b_j$. For example, we have that:
(i) $(1-2X_1 < 0)\ \precsim_M\ (3+X_1 < 0)$,
(ii) $(2-2X_1+X_2 < 0)\ \precsim_M\ (1+3X_1 < 0)$, and
(iii) $(1+3X_1 < 0)\ \not\precsim_M\ (2-2X_1+X_2 < 0)$.

Definition 3 (Generalization Operator $\ominus$). Let $\precsim$ be a thin wqo on the set $\mathcal{C}$ of constraints. A function $\ominus$ from $\mathcal{C}\times\mathcal{C}$ to $\mathcal{C}$ is a *generalization operator* with respect to $\precsim$ if, for all constraints c and d, we have: (i) $d \sqsubseteq c \ominus d$, and (ii) $c \ominus d \precsim c$.

A trivial generalization operator is defined as $c \ominus d = \mathit{true}$, for all constraints c and d (without loss of generality we assume that $\mathit{true} \precsim c$ for every constraint c). This operator is used during Phase (1) of our Software Model Checking method.

Definition 3 generalizes several operators proposed in the literature, such as the widening operator [4] and the *most specific generalization* operator [23,33].

Other generalization operators defined in terms of relations and operators on constraints such as *widening* and *convex-hull*, have been defined in [11]. Some of them can be found in Appendix A.

Now we describe a method for deriving, from any given generalization operator $\ominus$, a new version of that operator, denoted $\ominus_{cns}$, which adds some extra constraints and still is a generalization operator. The operator $\ominus_{cns}$ is called the *constrained generalization operator derived from* $\ominus$. Constrained generalization operators are used during Phase (2) of our Software Model Checking method.

In order to specify the constrained generalization operator we need the following notions.

Let $P \cup \{\gamma_0\}$ be the input program of the *Specialize* procedure. For any constraint d and atom A, we define the *unfeasible clauses* for the pair (d, A), denoted $\mathit{UnfCl}(d, A)$, to be the set $\{(H_1 \leftarrow c_1, G_1), \ldots, (H_m \leftarrow c_m, G_m)\}$, of (renamed apart) clauses of $P \cup \{\gamma_0\}$ such that, for $i = 1, \ldots, m$, A and H_i are unifiable via the most general unifier ϑ_i and $(d \wedge c_i)\,\vartheta_i$ is unsatisfiable.

The *head constraint* of a clause γ of the form $H \leftarrow c, A$ is the constraint $\mathit{project}(c, X)$, where X is the tuple of variables occurring in H. For any atomic constraint a, $\mathit{neg}(a)$ denotes the negation of a defined as follows: $\mathit{neg}(p < 0)$ is $-p \leq 0$ and $\mathit{neg}(p \leq 0)$ is $-p < 0$. Given a set C of clauses, we define the set of the *negated regions* of C, denoted $\mathit{NegReg}(C)$, as follows:

$$\mathit{NegReg}(C) = \{\mathit{neg}(a) \mid a \text{ is an atomic constraint of a head constraint of a clause in } C\}.$$

For any constraint d and atom A, we define the following constraint:

$$\mathit{cns}(d, A) = \bigwedge\{r \mid r \in \mathit{NegReg}(\mathit{UnfCl}(d, A)) \ \wedge\ d \sqsubseteq r\}.$$

We have that $d \sqsubseteq \mathit{cns}(d, A)$. Now, let $\ominus$ be a generalization operator with respect to the thin wqo $\precsim$. We define the constrained generalization operator derived from $\ominus$, as follows:

$$\ominus_{cns}(c, d, A) = (c \ominus d) \wedge \mathit{cns}(d, A).$$

Now we show that $\ominus_{cns}$ is indeed a generalization operator w.r.t. the thin wqo $\precsim_B$ we now define. Given a finite set B of (non necessarily atomic) constraints, a constraint $c_1 \wedge \ldots \wedge c_n$, where $c_1, \ldots, c_n$ are atomic, and a constraint d, we define the binary relation $\precsim_B$ on constraints as follows: $c_1 \wedge \ldots \wedge c_n \precsim_B d$ iff *either* (i) $(c_1 \wedge \ldots \wedge c_n) \precsim d$, *or* (ii) there exists $i \in \{1, \ldots, n\}$ such that $c_i \in B$ and $(c_1 \wedge \ldots \wedge c_{i-1} \wedge c_{i+1} \wedge \ldots \wedge c_n) \precsim_B d$. It can be shown that $\precsim_B$ is a thin wqo.

We observe that, for all constraints c, d, and all atoms A: (i) since $d \sqsubseteq c \ominus d$ and $d \sqsubseteq cns(d, A)$, then also $d \sqsubseteq \ominus_{cns}(c, d, A)$, and (ii) by definition of $\precsim_B$, for all constraints e, if $c \ominus d \precsim e$, then $\ominus_{cns}(c, d, A) \precsim_B e$, where $B = NegReg(P \cup \{\gamma_0\})$.

Thus, we have the following result.

Proposition 1. *For any program $P \cup \{\gamma_0\}$ given as input to the Specialize procedure, for any atom A, the operator $\ominus_{cns}(\_, \_, A)$ is a generalization operator with respect to the thin well-quasi ordering $\precsim_B$, where $B = NegReg(P \cup \{\gamma_0\})$.*

During Phase (2) in the *Specialize* procedure we use the following subprocedure *Generalize&Fold* which is an adaptation of the one in [11].

Procedure *Generalize&Fold*

Input: (i) a set *Defs* of definitions structured as a tree of definitions, called *DefsTree*, (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure, and (iii) a constrained generalization operator $\ominus_{cns}$.

Output: (i) A set *NewDefs* of new definitions, and (ii) a set Δ of folded clauses.

```
$NewDefs := \emptyset$;    $\Delta := \Gamma$;
while  in $\Delta$ there exists a clause $\delta$: $H \leftarrow d, G_1, A, G_2$ where the predicate symbol
of $A$ occurs in the body of some clause in $\Gamma$  do
GENERALIZE:
  Let $X$ be the set of variables occurring in $A$ and $d_X = project(d, X)$.
  1. if in $Defs \cup NewDefs$ there exists a (renamed apart) clause
        $\eta$: $newp(X) \leftarrow e, A$ such that $d_X \sqsubseteq e$ and $e \sqsubseteq cns(d_X, A)$
     then $NewDefs := NewDefs$
  2. elseif there exists a clause $\alpha$ in $Defs$ such that:
     (i) $\alpha$ is of the form $newq(X) \leftarrow b, A$, and (ii) $\alpha$ is the most recent ancestor
      of $\gamma$ in $DefsTree$ whose body contains a variant of $A$
     then $NewDefs := NewDefs \cup \{newp(X) \leftarrow \ominus_{cns}(b, d_X, A), A\}$
  3. else  $NewDefs := NewDefs \cup \{newp(X) \leftarrow d_X, A\}$
FOLD:
  $\Delta := (\Delta - \{\delta\}) \cup \{H \leftarrow d, G_1, newp(X), G_2\}$
end-while
```

The proof of termination of the *Specialize* procedure of Section 4 is a variant of the proof of Theorem 3 in [10]. In this variant we use Proposition 1 and we also take into account the fact that during the execution of the procedure, only a finite number of atoms are generated (modulo variants). Since the correctness of the *Specialize* procedure directly follows from the fact that the transformation rules preserve the least model semantics [7], we have the following result.

Theorem 1 (Termination and Correctness of Specialization). (i) *The Specialize procedure always terminates.* (ii) *Let program P_S be the output of the Specialize procedure. Then* `unsafe` $\in M(P)$ *iff* `unsafe` $\in M(P_S)$.

6 Experimental Evaluation

In this section we present some preliminary results obtained by applying our Software Model Checking method to some benchmark programs taken from the literature. The results show that our approach is viable and competitive with the state-of-the-art software model checkers.

Programs $ex1$, $f1a$, $f2$, and *interp* have been taken from the benchmark set of DAGGER [14]. Programs *substring* and *tracerP* are taken from [20] and [16], respectively. Programs $re1$ and *singleLoop* have been introduced to illustrate the constrained generalization strategy. Finally, *selectSort* is an encoding of the Selection sort algorithm where references to arrays have been replaced by using the nondeterministic choice operator `nd` to perform array bounds checking. The source code of all the above programs is available at `http://map.uniroma2.it/smc/`.

Our model checker uses the MAP system [26] which is a tool for transforming constraint logic programs implemented in SICStus Prolog. MAP uses the `clpr` library to operate on constraints over the reals. Our model checker consists of three modules: (i) a translator which takes a property *initProp* and a command *com* and returns their associated terms, (ii) the MAP system for CLP program specialization which performs Phases (1) and (2) of our method, and (iii) a program for computing the least models of CLP programs which performs Phase (3) of our method.

We have also run three state-of-the-art CLP-based software model checkers on the same set of programs, and we have compared their performance with that of our model checker. In particular, we have used: (i) ARMC [28], (ii) HSF(C) [13], and (iii) TRACER [17]. ARMC and HSF(C) are CLP-based software model checkers which implement the CEGAR technique. TRACER is a CLP-based model checker which uses Symbolic Execution (SE) for the verification of safety properties of sequential C programs using approximated preconditions or approximated postconditions.

Program	MAP				ARMC	HSF(C)	TRACER	
	W	W_{cns}	$CHWM$	$CHWM_{cns}$			$SPost$	$WPre$
$ex1$	1.08	1.09	1.14	1.25	0.18	0.21	∞	1.29
$f1a$	∞	∞	0.35	0.36	∞	0.20	$\perp$	1.30
$f2$	∞	∞	0.75	0.88	∞	0.19	∞	1.32
interp	0.29	0.29	0.32	0.44	0.13	0.18	∞	1.22
$re1$	∞	0.33	0.33	0.33	∞	0.19	∞	∞
selectSort	4.34	4.70	4.59	5.57	0.48	0.25	∞	∞
singleLoop	∞	∞	∞	0.26	∞	∞	$\perp$	1.28
substring	88.20	171.20	5.21	5.92	931.02	1.08	187.91	184.09
tracerP	0.11	0.12	0.11	0.12	∞	∞	1.15	1.28

Table 1. Time (in seconds) taken for performing model checking. '∞' means 'no answer within 20 minutes', and '$\perp$' means 'termination with error'.

Table 1 reports the results of our experimental evaluation which has been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

In Columns W and *CHWM* we report the results obtained by the MAP system when using the procedure presented in Section 5 and the generalization operators *Widen* and *CHWidenMax* [11], respectively. In Columns W_{cns} and $CHWM_{cns}$ we report the results for the constrained versions of those generalization operators, called $Widen_{cns}$ and $CHWidenMax_{cns}$, respectively. In the remaining columns we report the results obtained by ARMC, HSF(C), and TRACER using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options, respectively. More details on the experimental results can be found in Appendix B.

On the selected set of examples, we have that the MAP system with the $CHWidenMax_{cns}$ is able to verify 9 properties out of 9, while the other tools do not exceed 7 properties. Also the verification time is generally comparable to that of the other tools, and it is not much greater than that of the fastest tools. Note that there are two examples (*re*1 and *singleLoop*) where constrained generalization operators based on widening and convex-hull are strictly more powerful than the corresponding operators which are not constrained.

We also observe that the use of a constrained generalization operator usually causes a very small increase of the verification time with respect to the non-constrained counterparts, thus making constrained generalization a promising technique that can be used in practice for software verification.

7 Related Work and Conclusions

The specialization-based software model checking technique presented in this paper is an extension of the technique for the verification of safety properties of infinite state reactive systems, encoded as CLP programs, presented in [9,11]. The main novelties of the present paper are that here we consider imperative sequential programs and we propose a new specialization strategy which has the objective of preserving, if possible, the branching behaviour of the definitions to be generalized.

The use of constraint logic programming and program specialization for verifying properties of imperative programs has also been proposed by [27]. In that paper, the interpreter of an imperative language is encoded as a CLP program. Then the interpreter is specialized with respect to a specific imperative program to obtain a residual program on which a static analyser for CLP programs is applied. Finally, the information gathered during this process is translated back in the form of invariants of the original imperative program. Our approach does not require static analysis of CLP and, instead, we discover program invariants *during* the specialization process by means of (constrained) generalization operators.

The idea of constrained generalization which has the objective of preserving the branching behaviour of a clause, is related to the technique for preserving *characteristic trees* while applying abstraction during partial deduction [24]. Indeed, a characteristic tree provides an abstract description of the tree generated by unfolding a given goal, and abstraction corresponds to generalization. However, the partial deduction technique considered in [24] is applied to ordinary logic programs (not CLP programs) and constraints such as equations and inequations on finite terms, are only used in an intermediate phase.

In order to prove that a program satisfies a given property, software model checking methods try to automatically construct a conservative model (that is, a property-preserving model) of the program such that, if the model satisfies the given property, then also does the actual program. In constructing such a model a software model checker may follow two dual approaches: either (i) it may start from a coarse model and then progressively refine it by incorporating new facts, or (ii) it may start from a concrete model and then progressively abstract away from it some irrelevant facts.

Our verification method follows the second approach. Given a program P, we model its computational behaviour as a CLP program (Phase 1) by using the interpreter of the language in which P is written. Then, the CLP program is specialized with respect to the property to be verified, by using constrained generalization operators which have the objective of preserving, if possible, the branching behaviour of the definitions to be generalized. In this way we may avoid loss of precision, and at the same time, we enforce the termination of the specialization process (Phase 2).

In order to get a conservative model of a program, different generalization operators have been introduced in the literature. In particular, in [2] the authors introduce the *bounded widen* operator $c\,\nabla_B\, d$, defined for any given constraint c and d and any set B of constraints. This operator, which improves the precision of the *widen* operator introduced in [4], has been applied in the verification of synchronous programs and linear hybrid systems. A similar operator $c\,\nabla_B\, d$, called *widening up to* B, has been introduced in [15]. In this operator the set B of constraints is statically computed once the system to be verified is given. There is also a version of that operator, called *interpolated widen*, in which the set B is dynamically computed [14] by using the interpolants which are derived during the counterexample analysis.

Similarly to [2,5,14,15], the main objective of the constrained generalization operators introduced in this paper is the improvement of precision during program specialization. In particular, this generalization operator, similar to the bounded widen operator, limits the possible generalizations on the basis of a set of constraints defined by the CLP program obtained as output of Phase 1. Since this set of constraints which limits the generalization depends on the output of Phase 1, our generalization is more flexible than the one presented in [2]. Moreover, our generalization operator is more general than the classical widening operator introduced in [4]. Indeed, we only require that the set of constraints which have a non-empty intersection with the generalized constraint $c \ominus d$, are entailed by d.

Now let us point out some advantages of the techniques for software model checking which, like ours, use methodologies based on program specialization.

(1) First of all, the approach based on specialization of interpreters provides a parametric, and thus flexible, technique for software model checking. Indeed, by following this approach, given a program P written in the programming language L, and a property φ written in a logic M, in order to verify that φ holds for P, first (i) we specify the interpreter I_L for L and we specify the semantics S_M of M (as a proof system or a satisfaction relation) in a suitable metalanguage, then (ii) we specialize the interpreter and the semantics with respect to P and φ, and finally (iii) we analyze the derived specialized program (by possibly applying program specialization again, as done in this paper).

The metalanguage we used in this paper for Step (i) is CLP in which we have specified both the interpreter and the reachability relation (which defines the semantics of the reachability formula to be verified).

These features make program specialization a suitable framework for software model checking because it can easily adapt to the changes of the syntax and the semantics of the programming languages under consideration and also to the different logics where the properties of interest are expressed.

(2) By applying suitable generalization operators we can make sure that specialization always terminates and produces an equivalent program with respect to the property of interest. Thus, we can apply a sequence of specializations, thereby refining the analysis to the desired degree of precision.

(3) Program specialization provides a uniform framework for program analysis. Indeed, as already mentioned, abstraction operators can be regarded as particular generalization operators and, moreover, specialization can be easily combined to other program transformation techniques, such as program slicing, dead code elimination, continuation passing transformation, and loop fusion.

(4) Finally, on a more technical side, program specialization can easily accommodate polyvariant analysis [31] by introducing several specialized predicate definitions corresponding to the same point of the program to be analyzed.

Our preliminary experimental results show that our approach is viable and competitive with state-of-the-art software model checkers such as ARMC [28], HSF(C) [13] and TRACER [17].

In order to further validate of our approach, we plan in the near future to perform experiments on a larger set of examples. In particular, in order to support a larger set of input specifications, we are currently working on rewriting our translator so that it can take CIL (C Intermediate Language) programs [25]. We also plan to extend our interpreter to deal with more sophisticated features of imperative languages such as arrays, pointers, and procedure calls.

Moreover, since our specialization-based method preserves the semantics of the original specification, we also plan to explore how our techniques can be effectively used in a preprocessing step before using existing state-of-the-art software model checkers for improving both their precision and their efficiency.

References

1. T. Ball and S. K. Rajamani. Boolean programs: a model and process for software analysis. MSR TR 2000-14, Microsoft Report, 2000.
2. N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and assertions. *Proc. CP'95*, LNCS 976, pages 589–623. Springer, 1995.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. *Proc. CAV'00*, pages 154–169. Springer, 2000.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. *Proc. POPL'77*, pages 238–252. ACM Press, 1977.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *Proc. POPL'78*, pages 84–96. ACM Press, 1978.
6. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.
7. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. *Proc. LOPSTR'00*, LNCS 2042, pages 125–146. Springer, 2001.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proc. VCL'01*, DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying infinite state systems by specializing constraint logic programs. R. 657, IASI-CNR, Rome, Italy, 2007.
11. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theo. Pract. Log. Pro.*, DOI: 10.1017/S1471068411000627, 2013.
12. J. P. Gallagher. Tutorial on specialisation of logic programs. *Proc. PEPM'93*, pages 88–98. ACM Press, 1993.
13. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. *Proc. TACAS'12*, LNCS 7214, pages 549–551. Springer, 2012.
14. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. *Proc. TACAS'08*, LNCS 4963, pages 443–458. Springer, 2008. `www.cfdvs.iitb.ac.in/~bhargav/dagger.php`
15. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11:157–185, 1997.
16. J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.
17. J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification, 2012. `paella.d1.comp.nus.edu.sg/tracer/`
18. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. *Proc. CP'09*, LNCS 5732, pages 454–469. Springer, 2009.
19. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.
20. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. *Proc. TACAS'06*, LNCS 3920, pages 459–473. Springer, 2006.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

22. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theo. Pract. Log. Pro.*, 2(4&5):461–515, 2002.
23. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
24. M. Leuschel and D. De Schreye. Constrained partial deduction. *Proc. WLP'97*, Munich, Germany, pages 116–126, 1997.
25. G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Proc. CC'02*, LNCS 2304, pages 209–265. Springer, 2002. `kerneis.github.com/cil/`
26. The MAP transformation system. `www.iasi.cnr.it/~proietti/system.html`
27. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. *Proc. SAS'98*, LNCS 1503, pages 246–261. Springer, 1998.
28. A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: *Proc. PADL'07*, LNCS 4354, pages 245–259. Springer, 2007.
29. C. J. Reynolds. *Theories of Programming Languages.* Cambridge Univ. Press, 1998.
30. H. Saïdi. Model checking guided abstraction and analysis. *Proc. SAS'00*, pages 377–396. Springer, 2000.
31. S. Scott and T. Wang. Polyvariant flow analysis with constrained types. *Proc. ESOP'00*, LNCS 1782, pages 382–396. Springer, 2000.
32. N. Sharygina, S. Tonetta, and A. Tsitovich. An abstraction refinement approach combining precise and approximated techniques. *Soft. Tools Techn. Transf.*, 14(1):1–14, Springer, 2012.
33. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. *Proc. ILPS'95*, pages 465–479. MIT Press, 1995.

Appendix A: Some generalization operators

Here we define some generalization operators which have been used in the experiments we have performed (see also [11]).

• (W) Given any two constraints $c \equiv a_1, \ldots, a_m$, and d, the operator *Widen*, denoted $\ominus_W$, returns the constraint $a_{i1}, \ldots, a_{ir}$, such that $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$. Thus, *Widen* returns all atomic constraints of c that are entailed by d (see [4] for a similar widening operator used in static program analysis). The operator $\ominus_W$ is a generalization operator w.r.t. the thin wqo $\precsim_M$.

• (WM) Given any two constraints $c \equiv a_1, \ldots, a_m$, and $d \equiv b_1, \ldots, b_n$, the operator *WidenMax*, denoted $\ominus_{WM}$, returns the conjunction $a_{i1}, \ldots, a_{ir}, b_{j1}, \ldots, b_{js}$, where: (i) $\{a_{i1}, \ldots, a_{ir}\} = \{a_h \mid 1 \leq h \leq m \text{ and } d \sqsubseteq a_h\}$, and (ii) $\{b_{j1}, \ldots, b_{js}\} = \{b_k \mid 1 \leq k \leq n \text{ and } b_k \precsim_M c\}$.

The operator *WidenMax* is a generalization operator w.r.t. the thin wqo $\precsim_M$. It is similar to *Widen* but, together with the atomic constraints of c that are entailed by d, it returns also the conjunction of a subset of the atomic constraints of d.

Next we define a generalization operator by using the *convex hull* operator, which is often used in static program analysis [5].

• (*CH*) The *convex hull* of two constraints c and d in $\mathcal{C}$, denoted by $ch(c, d)$, is the least (w.r.t. the $\sqsubseteq$ ordering) constraint h in $\mathcal{C}$ such that $c \sqsubseteq h$ and $d \sqsubseteq h$. (Note that $ch(c, d)$ is unique up to equivalence of constraints.)

• (*CHWM*) Given any two constraints c and d, we define the operator *CHWiden-Max*, denoted $\ominus_{CHWM}$, as follows: $c \ominus_{CHWM} d = c \ominus_{WM} ch(c, d)$. The operator $\ominus_{CHWM}$ is a generalization operator w.r.t. the thin wqo $\precsim_M$.

CHWidenMax returns the conjunction of a subset of the atomic constraints of c and a subset of the atomic constraints of $ch(c, d)$.

Appendix B: Detailed experimental results

In Table 2 we present in some more detail the time taken for proving the properties of interest by using our method for software model checking with the generalization operators *Widen* (Column *W*) and *CHWidenMax* (Column *CHWM*) [11], and the constrained generalization operators derived from them $Widen_{cns}$ (Column W_{cns}) and $CHWidenMax_{cns}$ (Column $CHWM_{cns}$), respectively.

Columns *Ph*1, *Ph*2, and *Ph*3 show the time required during Phases (1), (2), and (3), respectively, of our Software Model Checking method presented in Section 3. The sum of these three times for each phase is reported in Column *Tot*.

Program		*W*			W_{cns}			*CHWM*			$CHWM_{cns}$		
	*Ph*1	*Ph*2	*Ph*3	*Tot*	*Ph*2	*Ph*3	*Tot*	*Ph*2	*Ph*3	*Tot*	*Ph*2	*Ph*3	*Tot*
*ex*1	1.02	0.05	0.01	1.08	0.07 ◁	0	1.09	0.11	0.01	1.14	0.23 ◁	0	1.25
*f*1*a*	0.35	0.01	∞	∞	0.01	∞	∞	0 ◁	0	0.35	0.01 ◁	0	0.36
*f*2	0.71	0.03	∞	∞	0.13	∞	∞	0.03 ◁	0.01	0.75	0.17 ◁	0	0.88
interp	0.27	0.01	0.01	0.29	0.02 ◁	0	0.29	0.04	0.01	0.32	0.17 ◁	0	0.44
*re*1	0.31	0.01	∞	∞	0.02 ◁	0	0.33	0.02 ◁	0	0.33	0.02 ◁	0	0.33
selectSort	4.27	0.06	0.01	4.34	0.43 ◁	0	4.70	0.3	0.02	4.59	1.3 ◁	0	5.57
singleLoop	0.22	0.02	∞	∞	0.02	∞	∞	0.03	∞	∞	0.04 ◁	0	0.26
substring	0.24	0.01	87.95	88.20	0.02	170.94	171.2	4.96 ◁	0.01	5.21	5.67 ◁	0.01	5.92
tracerP	0.11	0 ◁	0	0.11	0.01 ◁	0	0.12	0 ◁	0	0.11	0.01 ◁	0	0.12

Table 2. Time (in seconds) taken for performing software model checking with the MAP system. '∞' means 'no answer within 20 minutes'. Times marked by '◁' are relative to the programs obtained after Phase (2) and have no constrained facts (thus, for those programs the times of Phase (3) are very small ($\leq 0.01\,s$)).